Northeastern University

DS 5220 – Lecture 7
Ensemble Methods

Roi Yehoshua

Agenda

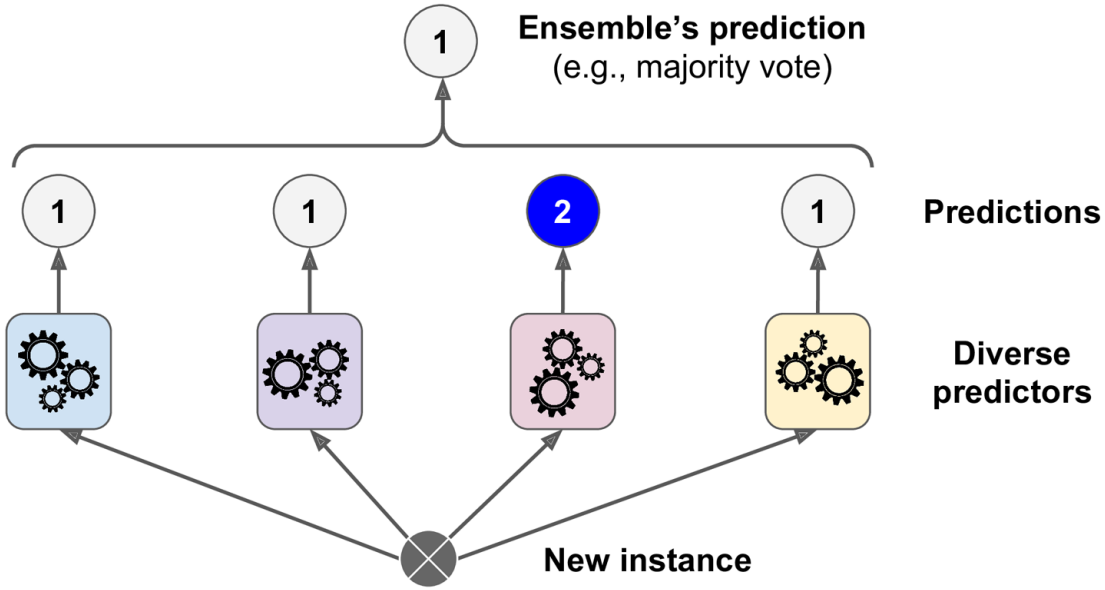
Northeastern University

- Ensemble learning
- Bagging methods
- Random forests
- Boosting methods
- AdaBoost
- Gradient boosting
- XGBoost

Ensemble Learning

Northeastern University

- Construct a set of **base estimators** from the training data
- Make a prediction on an unseen sample by combining their predictions
  - In classification by taking a vote on the classifiers' predictions (e.g., majority vote)
  - In regression by taking the average prediction of the regressors
- Improves generalization and robustness over a single estimator



Rationale for Ensemble Methods

Northeastern University

- Consider an ensemble of *n* base classifiers, each of which has an error rate ε
- The ensemble takes a majority vote on the predictions made by the base classifiers
- If the base classifiers are identical, then the ensemble's error rate will be also ε
- However, if the base classifiers are independent (their errors are uncorrelated), then the ensemble makes a wrong prediction only if more than half of them are wrong
- Therefore, the error rate of the ensemble is:

$$\epsilon_{\text{ensemble}} = \sum_{k=\lceil n/2 \rceil}^{n} \binom{n}{k} \epsilon^k (1-\epsilon)^{n-k}$$

- For example, if we have 25 base classifiers, each with an error rate of 0.25, then:

$$\epsilon_{\text{ensemble}} = \sum_{k=13}^{25} \binom{25}{k} 0.25^k 0.75^{25-k} = 0.0034 \ll 0.25$$

Rationale for Ensemble Methods

Northeastern University

- The error rate of an ensemble of 25 classifiers for different base classifier error rates:

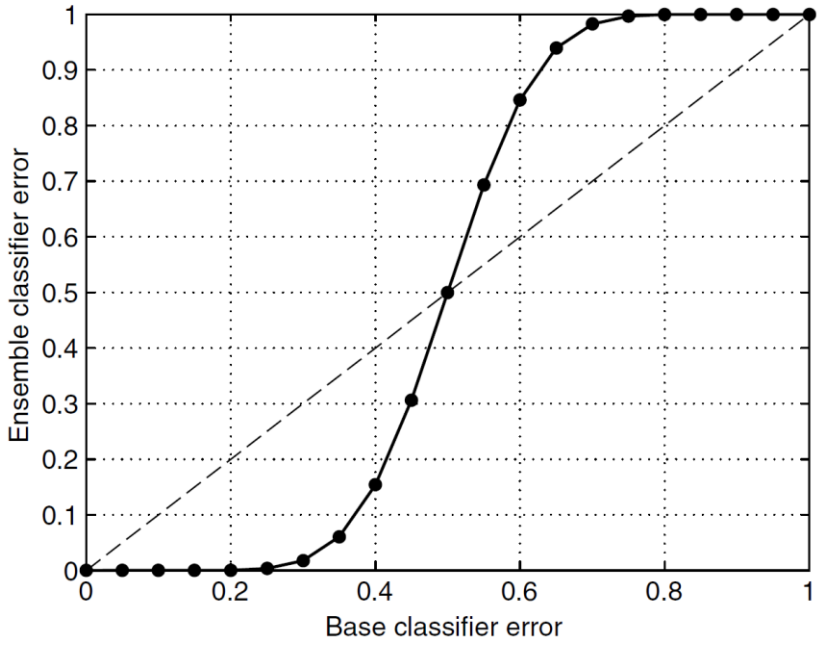


Figure 5.30. Comparison between errors of base classifiers and errors of the ensemble classifier.

Rationale for Ensemble Method

Northeastern University

- Two necessary conditions for an ensemble to perform better than a single classifier:
  - The base classifiers should be independent of each other
  - The base classifiers should be better than a classifier that performs random guessing

- Even if each classifier is a *weak learner* (i.e., it does only slightly better than random guessing), the ensemble can still be a *strong learner* (achieving high accuracy)

- In practice, it is difficult to ensure total independence among the base classifiers
- Nevertheless, improvement in accuracy can be obtained even when the base classifiers are slightly correlated

Rationale for Ensemble Methods

Northeastern University

- Ensemble methods also allows us to expand the hypothesis space
- For example, by combining linear classifiers we can obtain a non-linear classifier

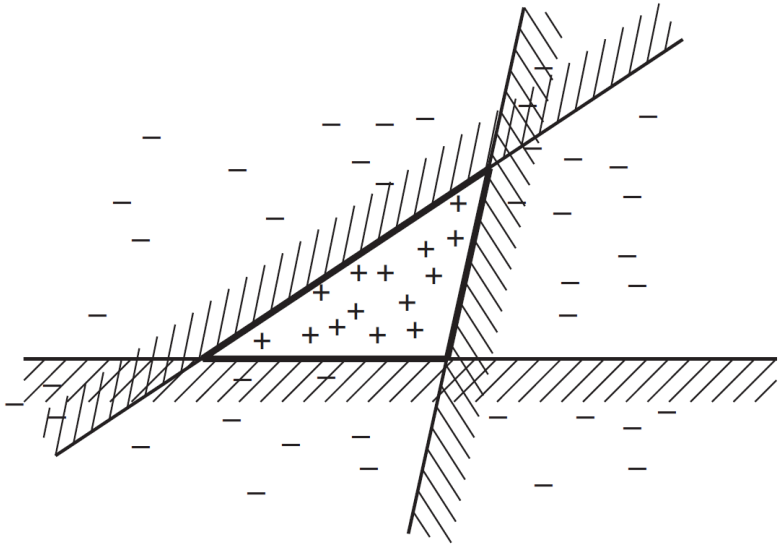


Figure 18.32 Illustration of the increased expressive power obtained by ensemble learning. We take three linear threshold hypotheses, each of which classifies positively on the unshaded side, and classify as positive any example classified positively by all three. The resulting triangular region is a hypothesis not expressible in the original hypothesis space.

How to Generate an Ensemble

Northeastern University

- Ensembles can be created in many ways:
  - By manipulating the learning algorithm
  - By manipulating the training set
  - By manipulating the input features

Algorithm 5.5 General procedure for ensemble method.

1: Let D denote the original training data, k denote the number of base classifiers, and T be the test data.
2: **for** $i = 1$ to k **do**
3: Create training set, D_i from D.
4: Build a base classifier C_i from D_i.
5: **end for**
6: **for** each test record $x \in T$ **do**
7: $C^*(x) = Vote(C_1(\mathbf{x}), C_2(\mathbf{x}), \ldots, C_k(\mathbf{x}))$
8: **end for**

Voting Classifier

Northeastern University

- Combine conceptually different ML classifiers and use a majority vote to predict the class labels
- Balance out the individual weaknesses of the different classifiers
- Two types of voting:
  - **Majority/Hard Voting** – returns the class label that represents the majority (mode) of the labels predicted by the base classifiers
  - **Soft Voting** – returns the class label as argmax of the sum of the predicted probabilities for the different classes
- Soft voting is possible only if all classifiers are able to estimate class probabilities
- Soft voting often achieves higher performance as it gives more weight to highly confident votes

Soft Voting Example

Northeastern University

- Assume that we have a 3-class classification problem and we use an ensemble of 3 base classifiers
- For a given sample, if the predicted probabilities of the classifiers are:

Classifier	Class 1	Class 2	Class 3
Classifier 1	0.2	0.5	0.3
Classifier 2	0.6	0.3	0.1
Classifier 3	0.3	0.4	0.3
Sum	**1.1**	**1.2**	**0.7**

- Then the predicted class label is 2, since it has the highest sum of predicted probabilities

Voting Classifiers in Scikit-Learn

Northeastern University

- The class VotingClassifier implements a majority rule/soft voting classifier

class `sklearn.ensemble.` **`VotingClassifier`**(*estimators*, *, *voting='hard'*, *weights=None*, *n_jobs=None*, *flatten_transform=True*, *verbose=False*) [source]

Argument	Description
estimators	A list of base classifiers. Invoking the fit() method on the VotingClassifier will fit clones of these estimators.
voting	'hard' or 'soft' (default = 'hard')
weights	Sequence of weights to weight the occurrences of predicted class labels (hard voting) or class probabilities before averaging (soft voting). Uses uniform weights if None.

- There is also an equivalent VotingRegressor class, which averages the predictions of several base regressors:

class `sklearn.ensemble.` **`VotingRegressor`**(*estimators*, *, *weights=None*, *n_jobs=None*, *verbose=False*) [source]

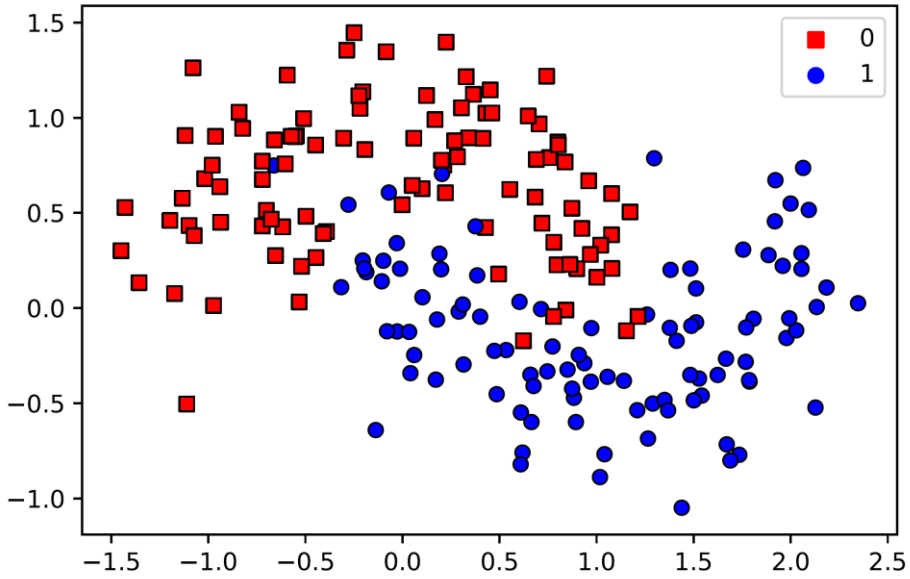
VotingClassifier Example

Northeastern University

- The following example shows how to fit a majority rule classifier, composed of three diverse classifiers, on the moons data set:

```
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=200, noise=0.25, random_state=0)
X_train, X_test, y_train, y_test = train_test_split(X, y)
```

```
sns.scatterplot(X[:, 0], X[:, 1], hue=y, style=y, markers=('s', 'o'),
                palette=('r', 'b'), edgecolor='black')
plt.savefig('Figures/moons.pdf')
```



VotingClassifier Example

Northeastern University

- We now define the base classifiers and the ensemble using a majority vote rule:

```
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import VotingClassifier

clf1 = LogisticRegression()
clf2 = KNeighborsClassifier()
clf3 = DecisionTreeClassifier()

ensemble = VotingClassifier(estimators=[('log', clf1), ('knn', clf2), ('dt', clf3)])
```

VotingClassifier Example

Northeastern University

- We'll use cross-validation to compare the base classifiers and the ensemble:

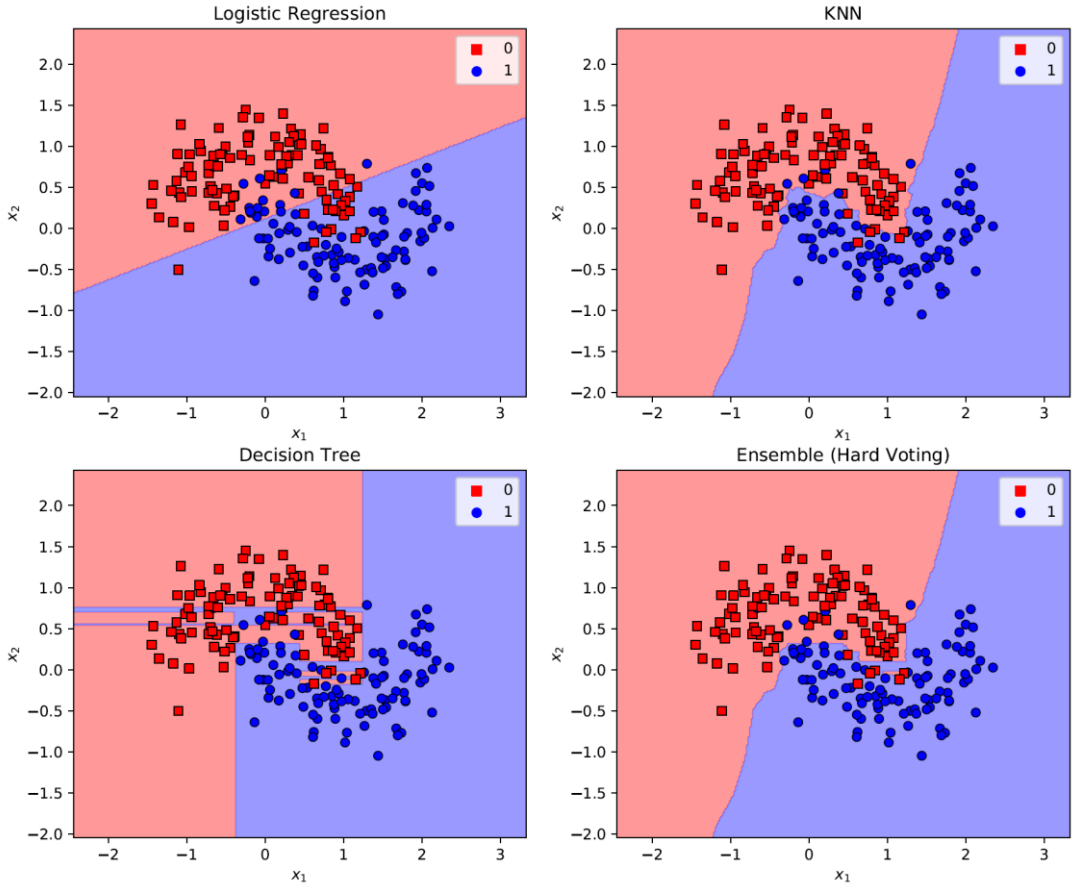
```
names = ['Logistic Regression', 'KNN', 'Decision Tree', 'Ensemble (Hard Voting)']
classifiers = [clf1, clf2, clf3, ensemble]

for clf, name in zip(classifiers, names):
    scores = cross_val_score(clf, X, y, cv=5)
    print(f'Accuracy: {scores.mean():.3f} (+/- {scores.std():.3f}) [{name}]')
```

```
Accuracy: 0.835 (+/- 0.051) [Logistic Regression]
Accuracy: 0.915 (+/- 0.049) [KNN]
Accuracy: 0.920 (+/- 0.019) [Decision Tree]
Accuracy: 0.925 (+/- 0.022) [Ensemble (Hard Voting)]
```

- The voting classifier slightly outperforms the base classifiers

VotingClassifier Example

- The decision regions of the ensemble seem to be a hybrid of the decision regions from the base classifiers:



Soft Voting

Northeastern University

- Using soft voting:

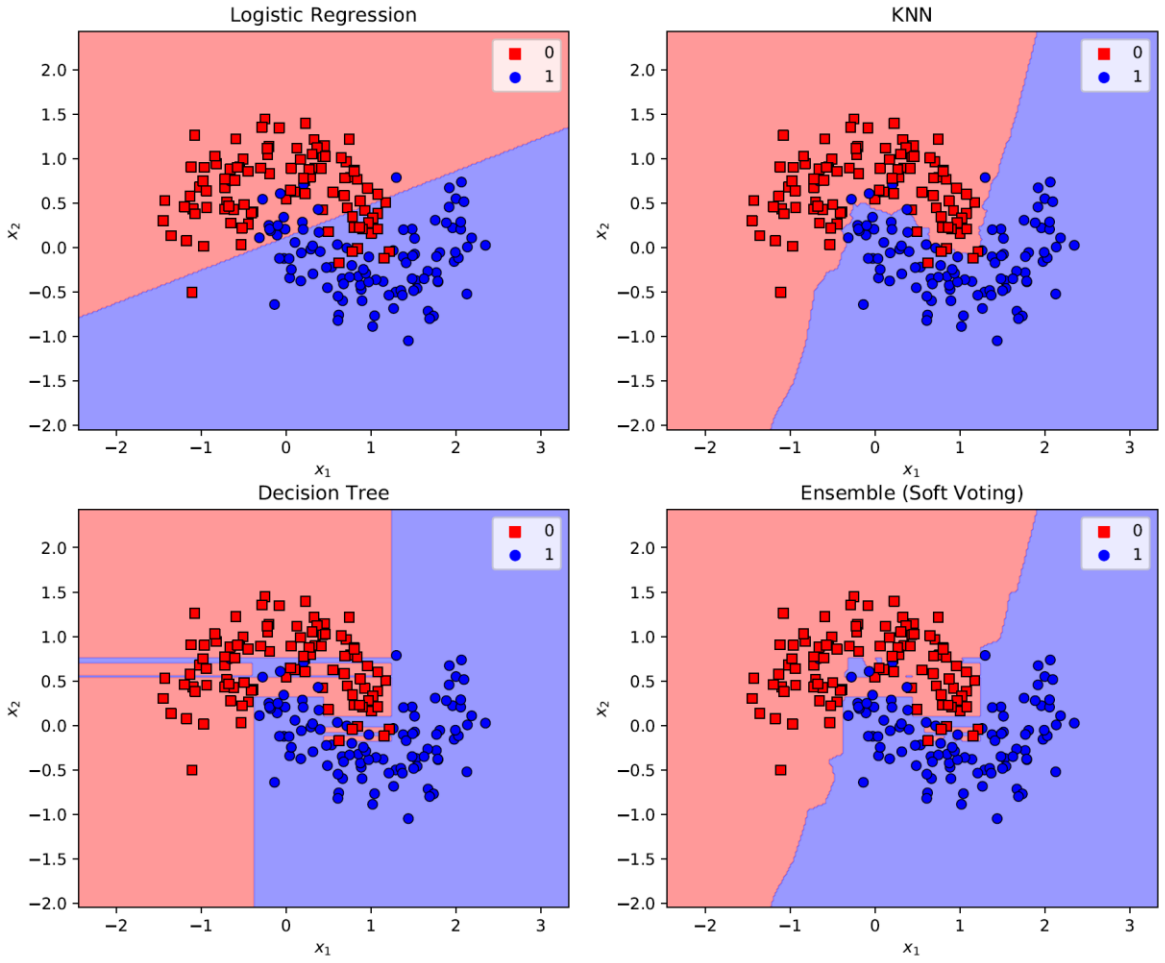
```
ensemble = VotingClassifier(estimators=[('log', clf1), ('knn', clf2), ('dt', clf3)],
                            voting='soft')
classifiers = [clf1, clf2, clf3, ensemble]
```

```
names = ['Logistic Regression', 'KNN', 'Decision Tree', 'Ensemble (Soft Voting)']
classifiers = [clf1, clf2, clf3, ensemble]

for clf, name in zip(classifiers, names):
    scores = cross_val_score(clf, X, y, cv=5)
    print(f'Accuracy: {scores.mean():.3f} (+/- {scores.std():.3f}) [{name}]')
```

```
Accuracy: 0.835 (+/- 0.051) [Logistic Regression]
Accuracy: 0.915 (+/- 0.049) [KNN]
Accuracy: 0.925 (+/- 0.027) [Decision Tree]
Accuracy: 0.930 (+/- 0.019) [Ensemble (Soft Voting)]
```

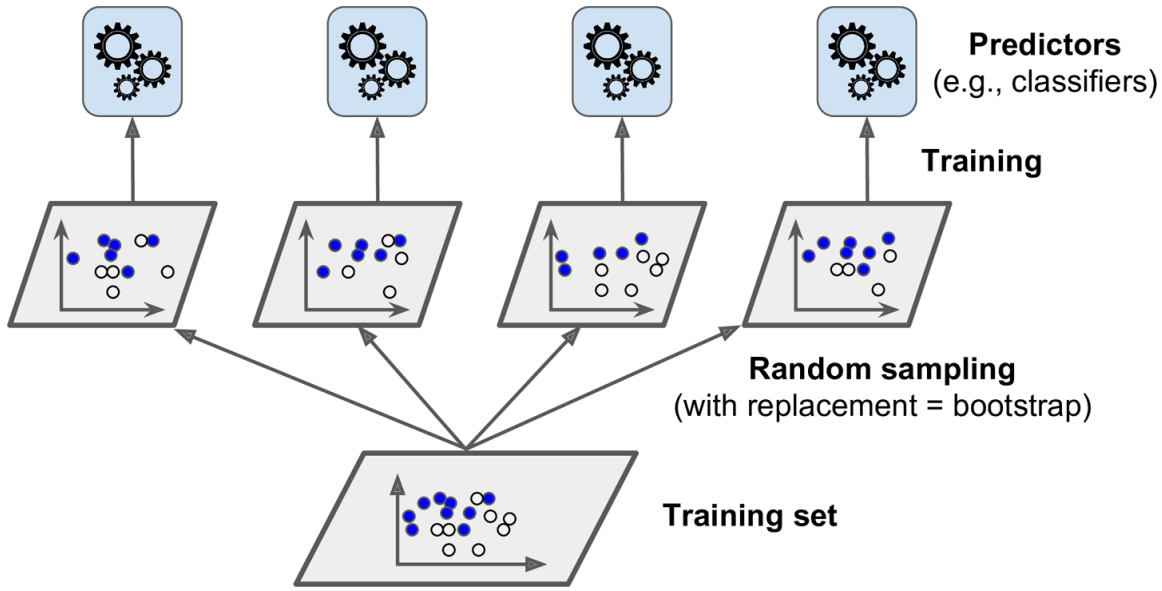
Soft Voting

- The decision boundaries of a soft voting classifier:



Bagging

- Bagging = bootstrap + aggregating
- Build *K* identical models on different random subsets of the training set
- Make a prediction by aggregating the predictions of the *K* models
  - e.g., by using majority voting



Bagging

Northeastern University

Algorithm 5.6 Bagging algorithm.

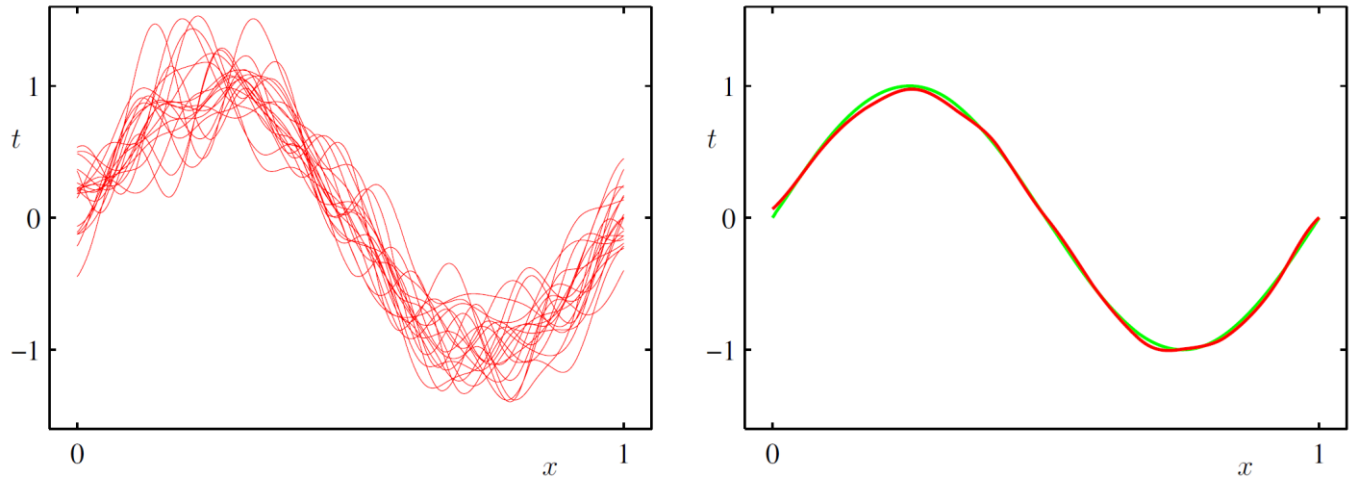
1: Let k be the number of bootstrap samples.
2: **for** $i = 1$ to k **do**
3: Create a bootstrap sample of size N, D_i.
4: Train a base classifier C_i on the bootstrap sample D_i.
5: **end for**
6: $C^*(x) = \underset{y}{\text{argmax}} \sum_i \delta\big(C_i(x) = y\big)$.

$\{\delta(\cdot) = 1$ if its argument is true and 0 otherwise$\}$.

Bagging

Northeastern University

- Bagging improves generalization error by reducing the variance of the base classifiers
  - The contributions from the variance term in each model tend to cancel each other
- For example, averaging 100 polynomials each trained on $n = 25$ data points generated from the sin function:



Bagging Methods

Northeastern University

- Bagging methods differ by the way they draw random subsets of the training set:
  - **Bagging** – sample random subsets of the training instances **with** replacement
  - **Pasting** – sample random subsets of the training instances **without** replacement
  - **Random Subspaces** – sample random subset of the features
    - This is useful when dealing with high-dimensional inputs such as images
  - **Random Patches** – sample random subsets of both the training instances and the features

- Bagging introduces a bit more diversity in the subsets the classifiers are trained on than pasting, which typically results in lower variance of the ensemble

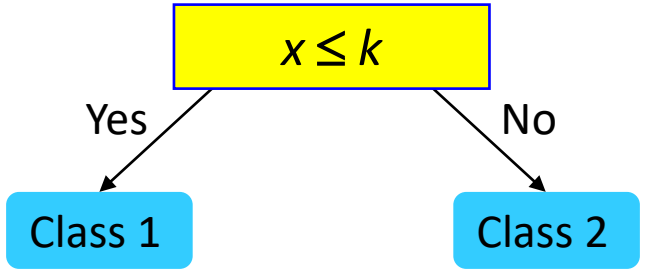
Bagging Example

Northeastern University

- Consider the following data set:

x	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
y	1	1	1	−1	−1	−1	−1	1	1	1

- Assume that our model is a one-level binary decision tree (aka **decision stump**)



- Without bagging, the best decision stump is the one that splits the samples at either $x \geq 0.35$ or $x \leq 0.75$
- Either way, the accuracy of the tree is at most 70%

Bagging Example

Northeastern University

- Suppose we apply the bagging procedure with the following 10 bootstrap samples:

Bagging Round 1:

x	0.1	0.2	0.2	0.3	0.4	0.4	0.5	0.6	0.9	0.9
y	1	1	1	1	-1	-1	-1	-1	1	1

x <= 0.35 ==> y = 1
x > 0.35 ==> y = -1

Bagging Round 2:

x	0.1	0.2	0.3	0.4	0.5	0.8	0.9	1	1	1
y	1	1	1	-1	-1	1	1	1	1	1

x <= 0.65 ==> y = 1
x > 0.65 ==> y = 1

Bagging Round 3:

x	0.1	0.2	0.3	0.4	0.4	0.5	0.7	0.7	0.8	0.9
y	1	1	1	-1	-1	-1	-1	-1	1	1

x <= 0.35 ==> y = 1
x > 0.35 ==> y = -1

Bagging Round 4:

x	0.1	0.1	0.2	0.4	0.4	0.5	0.5	0.7	0.8	0.9
y	1	1	1	-1	-1	-1	-1	-1	1	1

x <= 0.3 ==> y = 1
x > 0.3 ==> y = -1

Bagging Round 5:

x	0.1	0.1	0.2	0.5	0.6	0.6	0.6	1	1	1
y	1	1	1	-1	-1	-1	-1	1	1	1

x <= 0.35 ==> y = 1
x > 0.35 ==> y = -1

Bagging Example

Northeastern University

Bagging Round 6:

x	0.2	0.4	0.5	0.6	0.7	0.7	0.7	0.8	0.9	1
y	1	-1	-1	-1	-1	-1	-1	1	1	1

x <= 0.75 ==> y = -1
x > 0.75 ==> y = 1

Bagging Round 7:

x	0.1	0.4	0.4	0.6	0.7	0.8	0.9	0.9	0.9	1
y	1	-1	-1	-1	-1	1	1	1	1	1

x <= 0.75 ==> y = -1
x > 0.75 ==> y = 1

Bagging Round 8:

x	0.1	0.2	0.5	0.5	0.5	0.7	0.7	0.8	0.9	1
y	1	1	-1	-1	-1	-1	-1	1	1	1

x <= 0.75 ==> y = -1
x > 0.75 ==> y = 1

Bagging Round 9:

x	0.1	0.3	0.4	0.4	0.6	0.7	0.7	0.8	1	1
y	1	1	-1	-1	-1	-1	-1	1	1	1

x <= 0.75 ==> y = -1
x > 0.75 ==> y = 1

Bagging Round 10:

x	0.1	0.1	0.1	0.1	0.3	0.3	0.8	0.8	0.9	0.9
y	1	1	1	1	1	1	1	1	1	1

x <= 0.05 ==> y = -1
x > 0.05 ==> y = 1

Bagging Example

Northeastern University

- We now classify the data set using a majority vote on the base classifiers predictions:

Round	x=0.1	x=0.2	x=0.3	x=0.4	x=0.5	x=0.6	x=0.7	x=0.8	x=0.9	x=1.0
1	1	1	1	-1	-1	-1	-1	-1	-1	-1
2	1	1	1	1	1	1	1	1	1	1
3	1	1	1	-1	-1	-1	-1	-1	-1	-1
4	1	1	1	-1	-1	-1	-1	-1	-1	-1
5	1	1	1	-1	-1	-1	-1	-1	-1	-1
6	-1	-1	-1	-1	-1	-1	-1	1	1	1
7	-1	-1	-1	-1	-1	-1	-1	1	1	1
8	-1	-1	-1	-1	-1	-1	-1	1	1	1
9	-1	-1	-1	-1	-1	-1	-1	1	1	1
10	1	1	1	1	1	1	1	1	1	1
Sum	2	2	2	-6	-6	-6	-6	2	2	2
Sign	1	1	1	-1	-1	-1	-1	1	1	1
True Class	1	1	1	-1	-1	-1	-1	1	1	1

- The ensemble perfectly classifies all the training examples

Bagging in Scikit-Learn

Northeastern University

- Bagging methods are offered as a unified **BaggingClassifier** meta-estimator
  - or **BaggingRegressor** for regression problems

class `sklearn.ensemble.` **`BaggingClassifier`**(*base_estimator=None, n_estimators=10, *, max_samples=1.0, max_features=1.0, bootstrap=True, bootstrap_features=False, oob_score=False, warm_start=False, n_jobs=None, random_state=None, verbose=0*)

[source]

Argument	Description
base_estimator	The base estimator to fit on random subsets of the dataset (default is DecisionTreeClassifier)
n_estimators	The number of base estimators in the ensemble
max_samples	The number of samples to draw from X to train each base estimator. If int, then draw max_samples samples. If float, then draw max_samples * X.shape[0] samples.
max_features	The number of features to draw from X to train each base estimator. If int, then draw max_features features. If float, then draw max_features * X.shape[1] features.
bootstrap	Whether samples are drawn with replacement
bootstrap_features	Whether features are drawn with replacement
oob_score	Whether to use out-of-bag samples to estimate the generalization error

Bagging in Scikit-Learn

Northeastern University

- Let's train an ensemble of 500 decision tree classifiers on the moons data set
- Each tree is trained on 25% of the training instances, randomly sampled from the training set with replacement

```
from sklearn.ensemble import BaggingClassifier

bag_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=500,
                            max_samples=0.25, n_jobs=-1)

bag_clf.fit(X_train, y_train)
print('Accuracy on test set:', bag_clf.score(X_test, y_test))
```

```
Accuracy on test set: 0.92
```

- This is slightly better than the accuracy of a single decision tree:

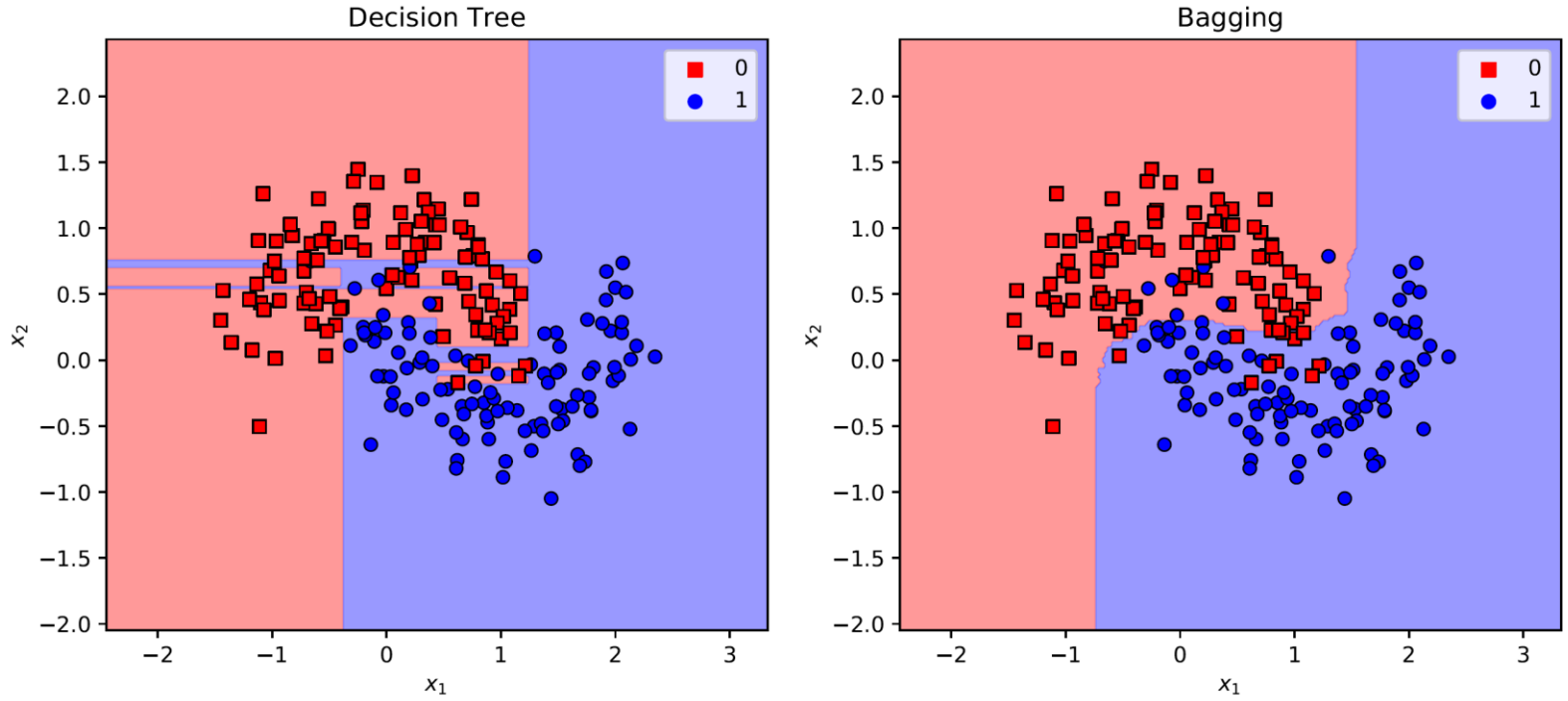
```
tree_clf = DecisionTreeClassifier()

tree_clf.fit(X_train, y_train)
print('Accuracy on test set:', tree_clf.score(X_test, y_test))
```

```
Accuracy on test set: 0.9
```

Bagging in Scikit-Learn

Northeastern University

- The decision boundary of the bagging ensemble vs. a single decision tree:



Out-of-Bag Evaluation

Northeastern University

- With bagging, some instances may be sampled several times, while others may not be sampled at all
- The probability of each data point of being selected at each bootstrap sample is

$$1 - (1 - 1/n)^n$$

  - where *n* is the number of training samples
- For large *n*, this probability converges to $1 - 1/e \approx 0.632$
- The remaining 37% of the training instances are called **out-of-bag** (OOB) samples
- Since a base classifier never sees the out-of-bag samples during training, they can be used as a validation set
- The ensemble itself can be evaluated by averaging out the out-of-bag evaluations of the base classifiers

Out-of-Bag Evaluation

Northeastern University

- In Scikit-Learn, you can set **oob_score=True** to request an automatic OOB evaluation
- The resulting evaluation score is available through the **oob_score_** variable:

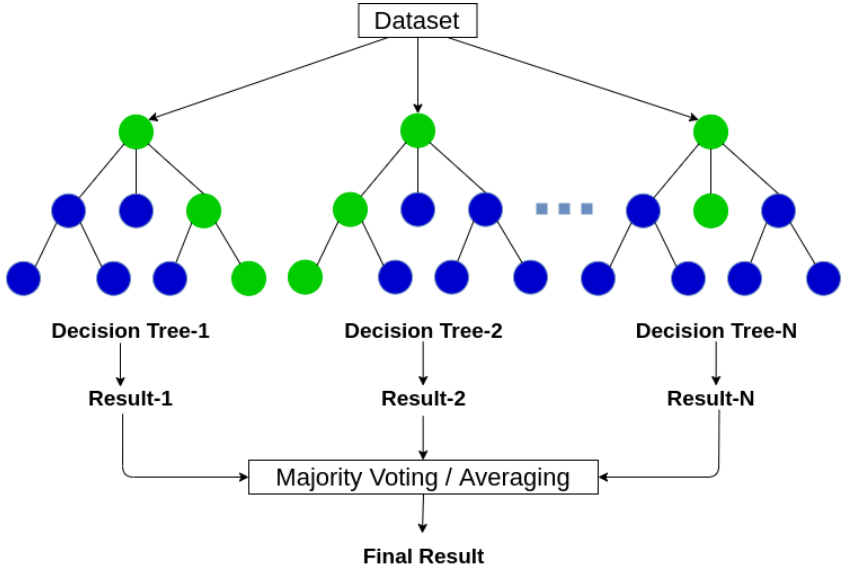
```
bag_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=500,
                            max_samples=0.25, n_jobs=-1, oob_score=True)

bag_clf.fit(X_train, y_train)
bag_clf.oob_score_
```

```
0.8933333333333333
```

Random Forests

Northeastern University

- A bagging method specifically designed for decision trees
- Each tree in the ensemble is built from a sample drawn with replacement
- Furthermore, when splitting each node during the construction of a tree, the best split is found from a random subset of the features
- This results in a greater tree diversity, which reduces the variance of the forest



Random Forests

Northeastern University

- It was theoretically proven (Breiman, 2001) that the upper bound on the generalization error of random forests converges to:

$$\epsilon_{ensemble} \leq \frac{\overline{\rho}(1-s^2)}{s^2}$$

  - $\overline{\rho}$ is the average correlation between the trees
  - *s* is a quantity that measures the "strength" of the tree classifiers in terms of their margin:

$$mr(\mathbf{x}, y) = P(h(\mathbf{x}) = y) - P(h(\mathbf{x}) \neq y)$$

$$s = \mathbb{E}_{\mathbf{x},y} mr(\mathbf{x}, y)$$

- As the trees become more correlated or the strength of the ensemble decreases, the generalization error bound tends to increase
- Randomization helps to reduce the correlation between the trees

Random Forests in Scikit-Learn

Northeastern University

- **RandomForestClassifier** implements a random forest for classification tasks:

> *class* `sklearn.ensemble.`**`RandomForestClassifier`**(*n_estimators=100, *, criterion='gini', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, bootstrap=True, oob_score=False, n_jobs=None, random_state=None, verbose=0, warm_start=False, class_weight=None, ccp_alpha=0.0, max_samples=None*) [source]

- It has all the parameters of a DecisionTreeClassifier (to control how trees are grown), plus the parameters of a BaggingClassifier to control the ensemble itself
- max_features determines the number of features to consider for each split:
  - If int, then consider max_features at each split
  - If float, then consider round(max_features * n_features) features at each split
  - If “auto”, then max_features = sqrt(n_features)
  - If “log2”, then max_features = log2(n_features)
  - If None, then max_features = n_features
- Similarly, there is a **RandomForestRegressor** class for regression tasks

Random Forests in Scikit-Learn

Northeastern University

- Training a random forest classifier with 500 trees on the moons dataset:

```
from sklearn.ensemble import RandomForestClassifier

forest_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1)

forest_clf.fit(X_train, y_train)
print('Accuracy on test set:', forest_clf.score(X_test, y_test))
```

```
Accuracy on test set: 0.94
```

Random Forests Summary

Northeastern University

Pros

- Often achieve higher accuracy than a single decision tree
- Less susceptible to overfitting
- More robust to noise

Cons

- Requires more computational resources than decision trees
- Not as easy to interpret
  - Following the paths of tens or hundreds of trees is much harder than a single tree
- Prediction is slower
- More hyperparameters to tune
  - e.g., the number of trees and the number of candidate features in each tree

Extra-Trees

Northeastern University

- In **extremely randomized trees** (or Extra-Trees), randomness goes one step further in the way splits are computed
- As in random forests, a random subset of candidate features is used for each split
- But instead of looking for the most discriminative thresholds (based on, e.g., information gain), a random threshold is selected for each candidate feature
  - The threshold is selected from a uniform distribution within the feature's values range
- The best of these randomly-generated thresholds is picked as the splitting rule
- This usually allows to reduce the variance of the model a bit more, at the expense of a slightly greater increase in bias
- Extra-Trees are also faster to train than regular random forests since finding the best possible threshold for each feature at every node is one of the most time-consuming tasks of growing a tree

Extra-Trees in Scikit-Learn

Northeastern University

- The ExtraTreesClassifier class implements an extra-trees classifier

> *class* `sklearn.ensemble.` **`ExtraTreesClassifier`**(*n_estimators=100, *, criterion='gini', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, bootstrap=False, oob_score=False, n_jobs=None, random_state=None, verbose=0, warm_start=False, class_weight=None, ccp_alpha=0.0, max_samples=None*) [source]

- Its API is identical to the RandomForestClassifier class
- Note that in random forests, bootstrap samples are used by default (bootstrap=True) while the default strategy for extra-trees is to use the whole dataset (bootstrap=False)
- Similarly, the **ExtraTreesRegressor** class has the same API as RandomForestRegressor

Extra-Trees in Scikit-Learn

Northeastern University

- Training an extra-trees classifier with 500 trees on the moons dataset:

```
from sklearn.ensemble import ExtraTreesClassifier

extra_clf = ExtraTreesClassifier(n_estimators=500, n_jobs=-1)

extra_clf.fit(X_train, y_train)
print('Accuracy on test set:', extra_clf.score(X_test, y_test))
```

```
Accuracy on test set: 0.96
```

- We've obtained slightly better results than the random forest classifier

Hyperparameter Tuning

Northeastern University

- The main parameters to adjust when using random forests and extra-trees are n_estimators and max_features
- For the number of trees in the forest:
  - The larger the better, but also the longer it will take to compute
  - Results will stop getting significantly better beyond a critical number of trees
- For the number of features to consider when splitting a node:
  - The lower the greater the reduction of variance, but also the greater the increase in bias
  - Empirical good default values are max_features="sqrt" for classification tasks, and max_features=None for regression problems
- Good results are often achieved when setting max_depth=None in combination with min_samples_split=2
- The best parameter values should always be cross-validated

Feature Importance Evaluation

Northeastern University

- The relative depth of a feature used as a decision node in a tree indicates its relative importance with respect to the predictability of the target variable
- Features used at the top of the tree contribute to the final prediction decision of a larger fraction of the samples
- In scikit-learn, the predictive power of each feature is estimated as a combination of the fraction of samples it contributes to and the decrease in impurity from splitting them
- By averaging these estimates over several randomized trees one can reduce the variance of such an estimate, and use it for **feature selection**

Feature Importance Evaluation

Northeastern University

- For example, the following code trains a RandomForestClassifier on the iris dataset and outputs the importance of each feature:

```
iris = load_iris()

clf = RandomForestClassifier(n_estimators=500, n_jobs=-1)
clf.fit(iris.data, iris.target)

for name, score in zip(iris.feature_names, clf.feature_importances_):
    print(f'{name}: {score:.4f}')
```

```
sepal length (cm): 0.0944
sepal width (cm): 0.0246
petal length (cm): 0.4358
petal width (cm): 0.4452
```

  - The most important features are petal length and width

Feature Importance Evaluation

Northeastern University

- Similarly, if you train a Random Forest classifier on the MNIST dataset and plot each pixel's importance, you get the following image:

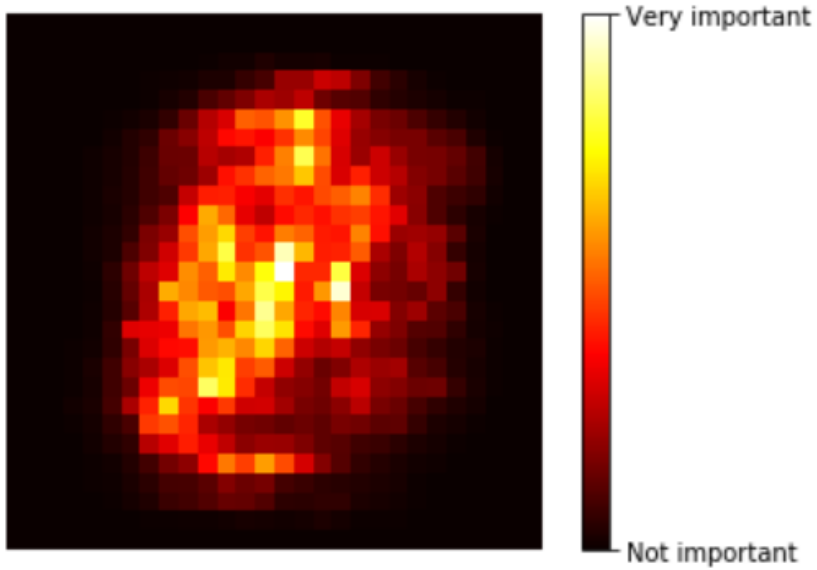
```
from scipy.io import loadmat
mnist_path = 'data/mnist-original.mat'
mnist = loadmat(mnist_path)
X = mnist['data'].T
y = mnist['label'][0]

train_size = 60000
X_train, y_train = X[:train_size], y[:train_size]
X_test, y_test = X[train_size:], y[train_size:]
```

```
rnd_clf = RandomForestClassifier(n_estimators=100, random_state=42)
rnd_clf.fit(X_train, y_train)
rnd_clf.score(X_test, y_test)
```

```
0.9688
```

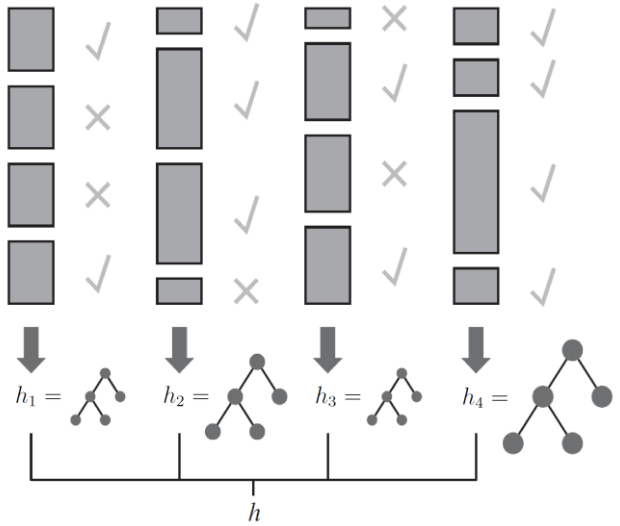
```
def plot_digit(data):
    image = data.reshape(28, 28)
    plt.imshow(image, cmap=plt.cm.hot)
    plt.axis('off')
```

```
plot_digit(rnd_clf.feature_importances_)
cbar = plt.colorbar(ticks=[rnd_clf.feature_importances_.min(),
                           rnd_clf.feature_importances_.max()])
cbar.ax.set_yticklabels(['Not important', 'Very important'])
plt.show()
```



Boosting

- In boosting, the base classifiers are trained sequentially
- Samples that are misclassified by a base classifier are given greater weight when used to train the next classifier
  - Each classifier is thereby focused on examples that were misclassified by the previous ones
- The predictions of the classifiers are combined through a weighted majority vote
  - The weights are based on how well the classifiers performed on the training set



Boosting Algorithms

Northeastern University

- Boosting algorithms differ by:
  - how the weights are updated at the end of each boosting round
  - how the predictions made by the base classifiers are combined
- Two popular algorithms:
  - AdaBoost
  - Gradient boosting

AdaBoost

Northeastern University

- A boosting algorithm introduced in 1995 by Yoav Freund and Robert Schapire, who won the 2003 Gödel Prize for their work
- AdaBoost can be used to boost the performance of any ML algorithm
- It is typically used with weak learners
- **Weak learner** – a classifier whose accuracy is only slightly better than random guessing (50% + ε)
  - Decision stumps are often used as weak learners
- If each of the base classifier is a weak learner, Adaboost is guaranteed to return a model that classifies the training data perfectly for a large enough rounds of boosting

AdaBoost

Northeastern University

- Suppose we have a data set $\{(x_i, y_i), i = 1, \ldots, n\}$, where $y_i \in \{-1, 1\}$
- Initially, we set the distribution of the samples to be uniform:

$$D_1(i) = \frac{1}{n} \quad i = 1, \ldots, n$$

- In each boosting round ($t = 1, \ldots, T$) we train a weak classifier using distribution D_t
  - Samples are drawn according to the sampling distribution
- The weak classifier outputs a hypothesis $h_t(x) \in \{-1, +1\}$
- The importance of each classifier depends on its error rate, defined as:

$$\epsilon_t = P_{i \sim D_t}[h_t(x_i) \neq y_i] = \sum_{i: h_t(x_i) \neq y_i} D_t(i)$$

AdaBoost

Northeastern University

- The importance of each base classifier h_t is given by the following parameter:

$$\alpha_t = \eta \log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$$

  - η is a learning rate hyperparameter (typically $\eta = 0.5$)
- α_t has a large positive value if the error rate is close to 0
- α_t has a large negative value if the error rate is close to 1
- If the classifier is just guessing randomly, α_t will be close to 0

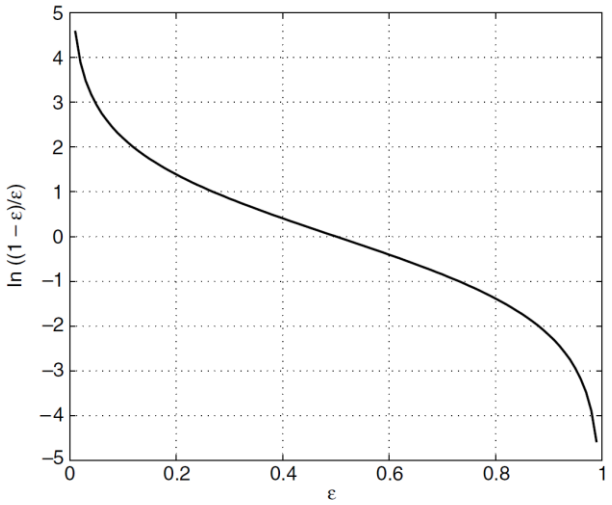


Figure 5.37. Plot of α as a function of training error ϵ.

AdaBoost

Northeastern University

- The α_t parameter is also used to update the weights of the training samples
- For each training sample *i* = 1, …, *n* we update:

$$D_{t+1}(i) = \frac{D_t(i)\exp(-\alpha_t y_i h_t(x_i))}{Z_t}$$

  - Z_t is a normalization factor used to ensure that D_{t+1} is a distribution, i.e.,

$$Z_t = \sum_{i=1}^{n} D_t(i)\exp(-\alpha_t y_i h_t(x_i))$$

- The distribution update formula increases the weight of incorrectly classified examples and decreases the weights of those classified correctly
  - Assuming $\alpha_t > 0$:

$$y_i = h_t(\mathbf{x}_i) \Rightarrow y_i h_t(\mathbf{x}_i) = 1 \Rightarrow \exp(-\alpha_t y_i h_t(\mathbf{x}_i)) = \exp(-\alpha_t) < 1$$

$$y_i \neq h_t(\mathbf{x}_i) \Rightarrow y_i h_t(\mathbf{x}_i) = -1 \Rightarrow \exp(\alpha_t y_i h_t(\mathbf{x}_i)) = \exp(\alpha_t) > 1$$

AdaBoost

Northeastern University

- The prediction made by each classifier h_t is weighted according to α_t
- Therefore, the final hypothesis is:

$$H(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$$

- This allows AdaBoost to penalize models that have poor accuracy
  - e.g., those generated at earlier boosting rounds

AdaBoost Algorithm

Northeastern University

Algorithm AdaBoost

Input:

Training set $\{(x_1, y_1), ..., (x_n, y_n)\}$, where $x_i \in X$, $y_i \in \{-1, +1\}$
a weak learner algorithm
number of boosting rounds T
learning rate η

1: Initialize $D_1(i) = 1/n$ for $i = 1, ..., n$.
2: **for** $t = 1, ..., T$ **do**:
3: Train a weak learner using distribution D_t
4: Get weak hypothesis $h_t : X \to \{-1, +1\}$ with error

$$\epsilon_t = \sum_{i:h_t(x_i) \neq y_i} D_t(i)$$

5: Choose

$$\alpha_t = \eta \log \left(\frac{1 - \epsilon_t}{\epsilon_t} \right)$$

6: Update, for $i = 1, ..., n$:

$$D_{t+1}(i) = \frac{D_t(i) \exp(-\alpha_t y_i h_t(x_i))}{Z_t}$$

where Z_t is a normalization factor.
7: Output the final hypothesis:

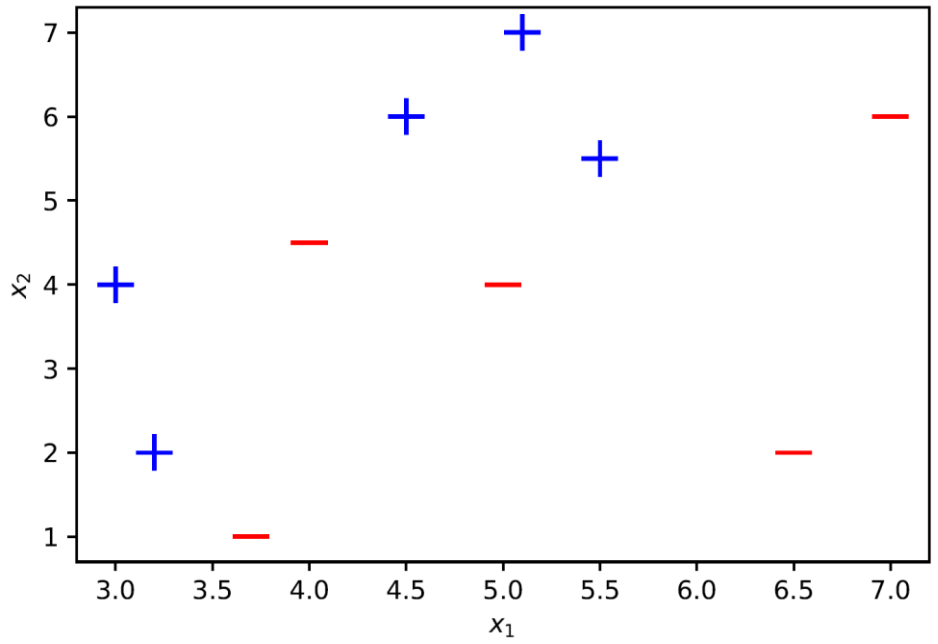
$$H(x) = \text{sign} \left(\sum_{t=1}^{T} \alpha_t h_t(x) \right)$$

AdaBoost Example

Northeastern University

- Let's examine how AdaBoost works on the following data set:

sample	x1	x2	class	weight
1	3	4	1	0.1
2	3.2	2	1	0.1
3	3.7	1	-1	0.1
4	4	4.5	-1	0.1
5	4.5	6	1	0.1
6	5	4	-1	0.1
7	5.1	7	1	0.1
8	5.5	5.5	1	0.1
9	6.5	2	-1	0.1
10	7	6	-1	0.1

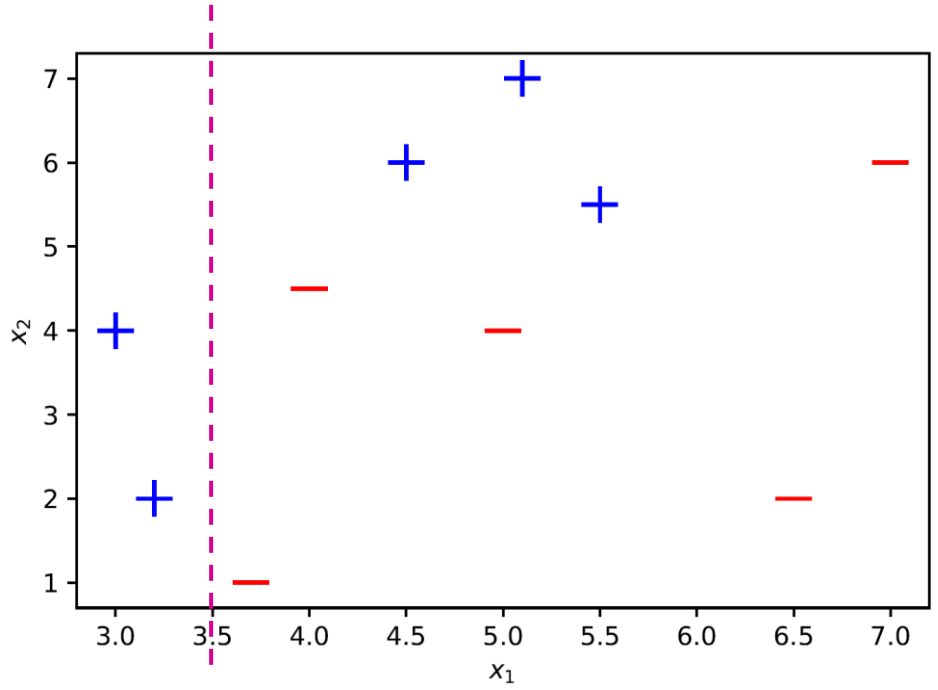


First Boosting Round

Northeastern University

- The distribution of weights:

sample	x1	x2	class	weight
1	3	4	1	0.1
2	3.2	2	1	0.1
3	3.7	1	-1	0.1
4	4	4.5	-1	0.1
5	4.5	6	1	0.1
6	5	4	-1	0.1
7	5.1	7	1	0.1
8	5.5	5.5	1	0.1
9	6.5	2	-1	0.1
10	7	6	-1	0.1



Best decision stump is: $x_1 \leq 3.5$

First Boosting Round

Northeastern University

- The error rate of the first decision stump is $\varepsilon = 3 \cdot 0.1 = 0.3$
- The importance factor is:

$$\alpha_1 = 0.5 \ln\left(\frac{1 - 0.3}{0.3}\right) = 0.424$$

- The change in weights is:

$$w_{incorrect} = 0.1 \cdot e^{0.424} = 0.153$$

$$w_{correct} = 0.1 \cdot e^{-0.424} = 0.065$$

$$Z_1 = 3 \cdot 0.153 + 7 \cdot 0.065 = 0.917$$

$$w_{incorrect} = \frac{0.153}{0.917} = 0.167$$

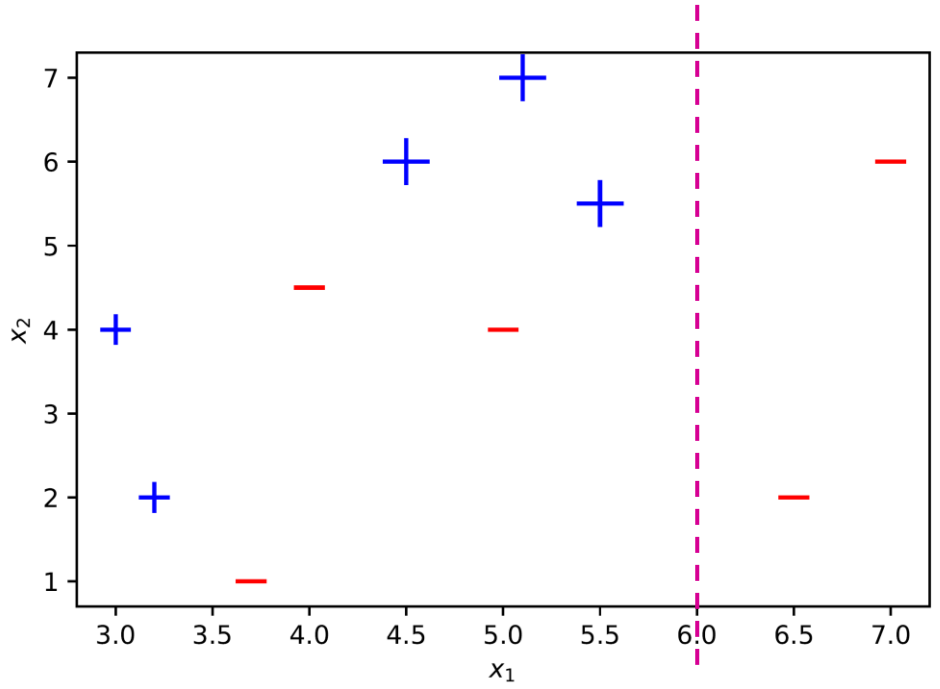
$$w_{correct} = \frac{0.065}{0.917} = 0.071$$

Second Boosting Round

Northeastern University

- The new distribution of weights:

sample	x1	x2	class	weight
1	3	4	1	0.071
2	3.2	2	1	0.071
3	3.7	1	-1	0.071
4	4	4.5	-1	0.071
5	4.5	6	1	0.167
6	5	4	-1	0.071
7	5.1	7	1	0.167
8	5.5	5.5	1	0.167
9	6.5	2	-1	0.071
10	7	6	-1	0.071



Best decision stump is: $x_1 \leq 6$

Second Boosting Round

Northeastern University

- The error rate of the second decision stump is $\varepsilon = 3 \cdot 0.071 = 0.213$
- The importance factor is:

$$\alpha_2 = 0.5 \ln\left(\frac{1 - 0.213}{0.213}\right) = 0.65$$

- The change in weights is:

$$w_{3,4,6} = 0.071 \cdot e^{0.65} = 0.136$$

$$w_{1,2,9,10} = 0.071 \cdot e^{-0.65} = 0.037$$

$$w_{5,7,8} = 0.167 \cdot e^{-0.65} = 0.087$$

$$Z_2 = 3 \cdot 0.136 + 4 \cdot 0.037 + 3 \cdot 0.087 = 0.817$$

$$w_{3,4,6} = \frac{0.136}{0.817} = 0.167$$

$$w_{1,2,9,10} = \frac{0.037}{0.817} = 0.045$$

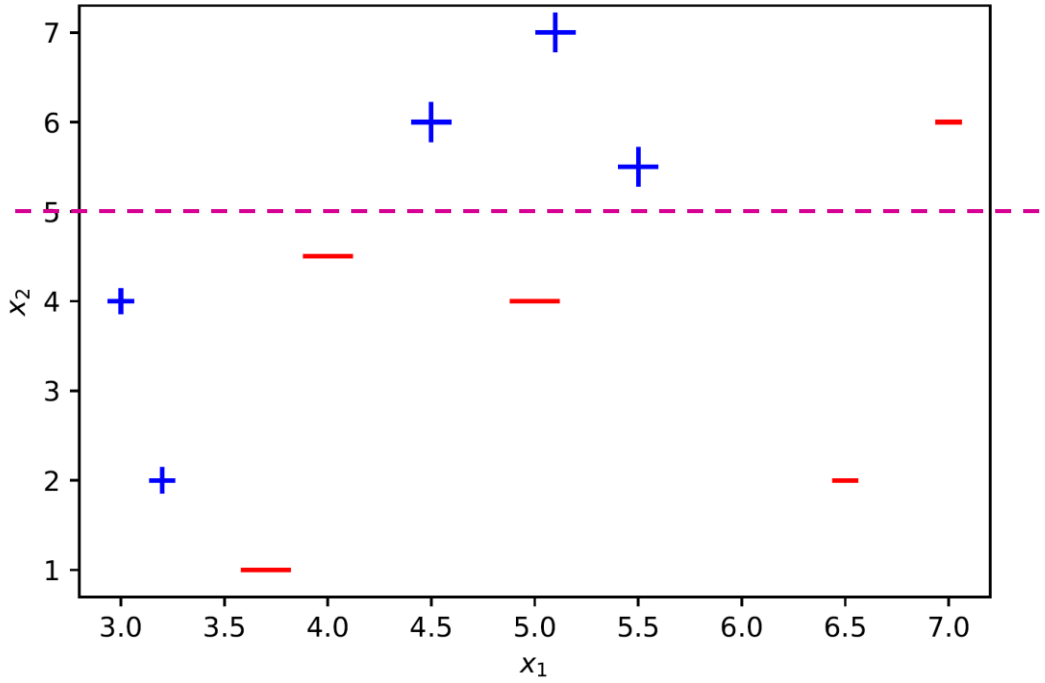
$$w_{5,7,8} = \frac{0.087}{0.817} = 0.106$$

Third Boosting Round

Northeastern University

- The new distribution of weights:

sample	x1	x2	class	weight
1	3	4	1	0.045
2	3.2	2	1	0.045
3	3.7	1	-1	0.167
4	4	4.5	-1	0.167
5	4.5	6	1	0.106
6	5	4	-1	0.167
7	5.1	7	1	0.106
8	5.5	5.5	1	0.106
9	6.5	2	-1	0.045
10	7	6	-1	0.045



Best decision stump is: $x_2 \geq 5$

Third Boosting Round

Northeastern University

- The error rate of the third decision stump is $\varepsilon = 3 \cdot 0.045 = 0.135$
- The importance factor is:

$$\alpha_3 = 0.5 \ln\left(\frac{1 - 0.135}{0.135}\right) = 0.923$$

Final Hypothesis

Northeastern University

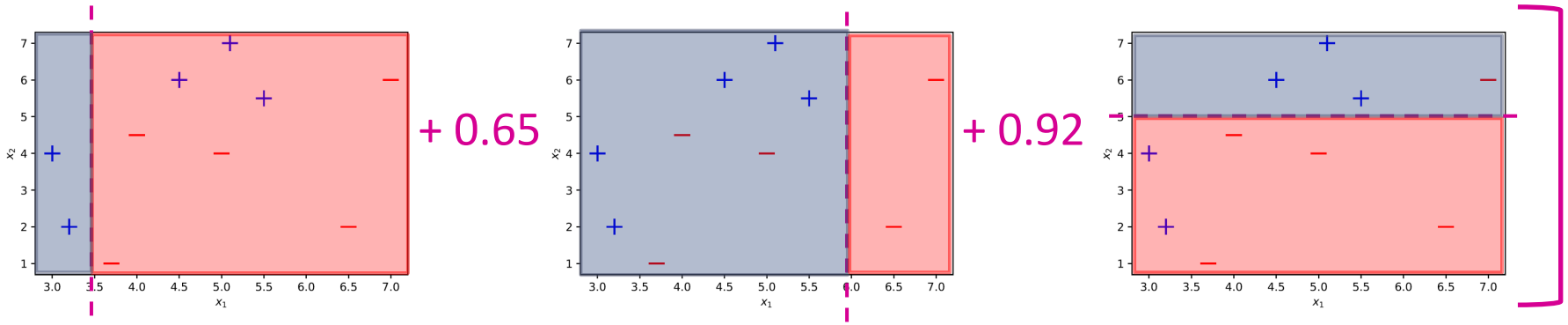
- The final prediction of the ensemble is obtained by taking a weighted average of the predictions made by each base classifier:

Round	1	2	3	4	5	6	7	8	9	10
1	1	1	-1	-1	-1	-1	-1	-1	-1	-1
2	1	1	1	1	1	1	1	1	-1	-1
3	-1	-1	-1	-1	1	-1	1	1	-1	1
Sum	0.15	0.15	-0.697	-0.697	1.149	-0.697	1.149	1.149	-1.996	-0.15
Sign	1	1	-1	-1	1	-1	1	1	-1	-1

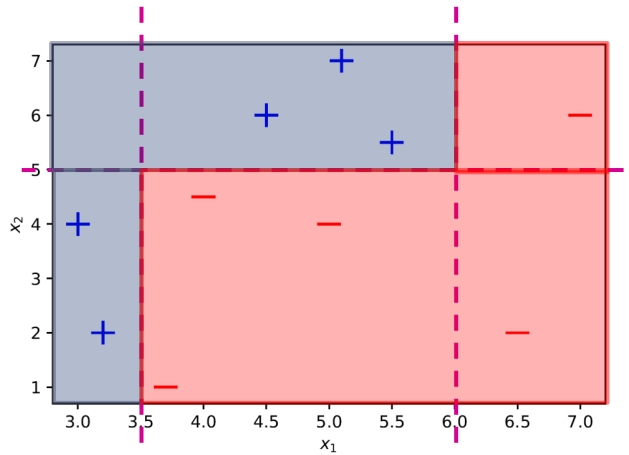
- The final ensemble perfectly classifies all examples in the training set

Final Hypothesis

$H_{\text{final}} = \text{sign}\left[0.42 \; + 0.65 \; + 0.92\right]$



$=$



Ensemble Error Rate

Northeastern University

- An important analytical result of boosting shows that the training error of the ensemble decreases exponentially
- The training error of the ensemble is bounded by:

$$\epsilon_{\text{ensemble}} \leq \prod_t 2\sqrt{\epsilon_t(1-\epsilon_t)}$$

You will prove this in the assignment!

- Let

$$\epsilon_t = 0.5 - \gamma_t$$

  - γ_t measures how much better the classifier is than random guessing
- If each weak classifier is slightly better than random, so $\gamma_k > \gamma$ for some $\gamma > 0$, then the training error drops exponentially fast in T:

$$\epsilon_{\text{ensemble}} \leq \prod_t \sqrt{1-4\gamma_t^2} \leq \exp\left(-2\sum_t \gamma_t^2\right) \leq \exp(-2T\gamma^2)$$

AdaBoost in Scikit-Learn

Northeastern University

- For classification, AdaBoostClassifier implements the AdaBoost-SAMME and AdaBoost-SAMME.R algorithms
  - These are extensions of the original AdaBoost to multi-class problems
  - SAMME.R utilizes class probabilities for the prediction if the base classifiers provide them
  - J. Zhu, H. Zou, S. Rosset, T. Hastie, “Multi-class AdaBoost”, 2009

> *class* `sklearn.ensemble.` **`AdaBoostClassifier`**(*base_estimator=None, *, n_estimators=50, learning_rate=1.0, algorithm='SAMME.R', random_state=None*) [source]

  - The default base estimator is DecisionTreeClassifier with max_depth=1
- For regression, AdaBoostRegressor implements AdaBoost.R2
  - H. Drucker, “Improving Regressors using Boosting Techniques”, 1997

> *class* `sklearn.ensemble.` **`AdaBoostRegressor`**(*base_estimator=None, *, n_estimators=50, learning_rate=1.0, loss='linear', random_state=None*) [source]

AdaBoost in Scikit-Learn

Northeastern University

- Training an AdaBoost classifier with 200 decision stumps on the moons data set:

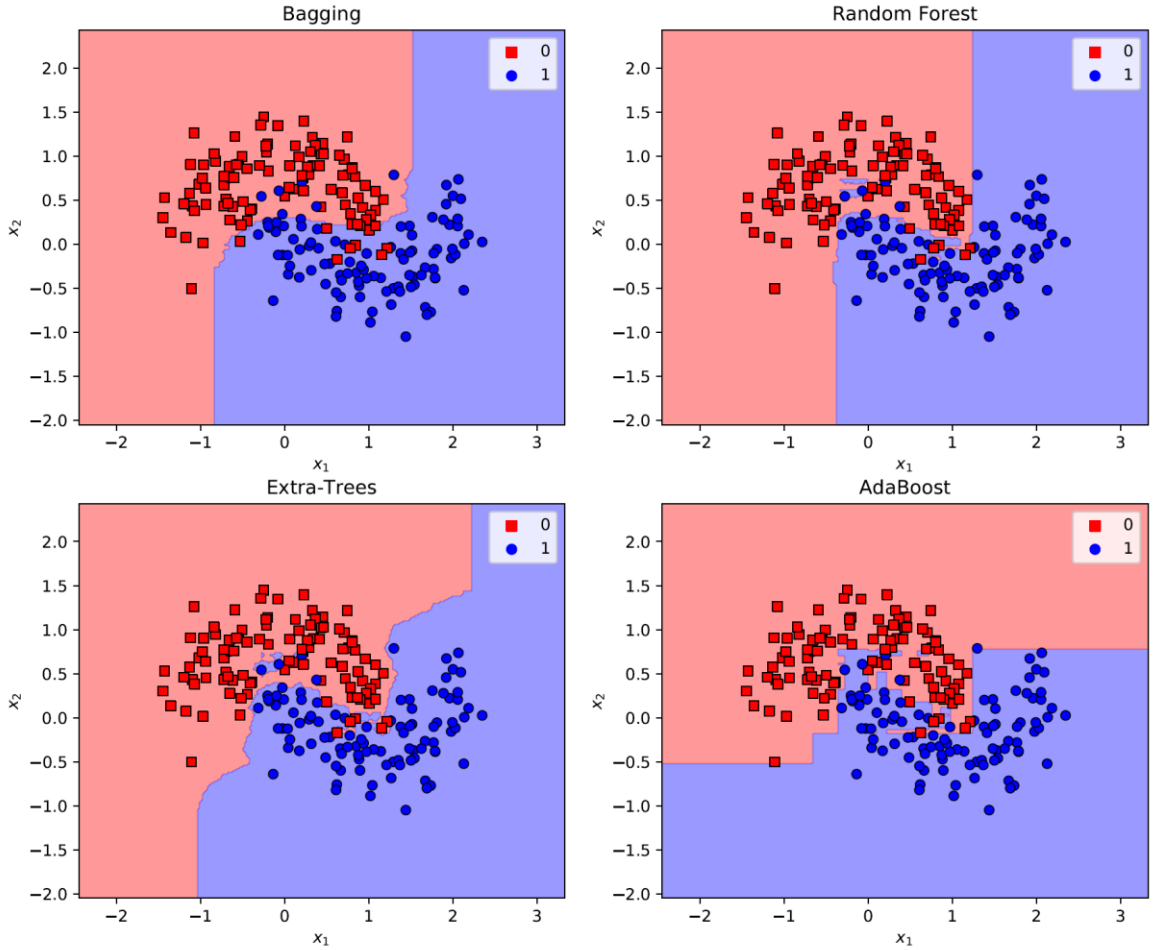
```
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(n_estimators=200)

ada_clf.fit(X_train, y_train)
print('Accuracy on test set:', ada_clf.score(X_test, y_test))
```

```
Accuracy on test set: 0.94
```

- If your AdaBoost ensemble is overfitting the training set, you can try reducing the number of estimators or more strongly regularizing the base estimator

Decision Boundaries

- The decision boundaries of the different ensembles:



Tuning the Hyperparameters

Northeastern University

- There is a trade-off between the learning rate and the number of weak classifiers
- A learning rate of η shrinks the contribution of each classifier (α_t) by η
- Decreasing the learning rate leads to smaller variations in the sample weights
  - This in turn slows down learning, which allows to reduce overfitting
- On the other hand, low values of η require more boosting iterations, in order to allow the weak classifier to continue to improve
- However, increasing the number of weak classifiers leads to more complex overall decision boundaries and thus increases overfitting

AdaBoost Summary

Northeastern University

Pros

- Achieves higher accuracy than a single decision tree, but comparable to accuracy of random forests
- Less prone to overfitting
- Training of each base classifier is fast
- Has small number of tunable parameters

Cons

- Sensitive to outliers, due to the exponential loss function
- Training cannot be parallelized

Ensemble Methods Comparison

Northeastern University

Data Set	Number of (Attributes, Classes, Records)	Decision Tree (%)	Bagging (%)	Boosting (%)	RF (%)
Anneal	(39, 6, 898)	92.09	94.43	95.43	95.43
Australia	(15, 2, 690)	85.51	87.10	85.22	85.80
Auto	(26, 7, 205)	81.95	85.37	85.37	84.39
Breast	(11, 2, 699)	95.14	96.42	97.28	96.14
Cleve	(14, 2, 303)	76.24	81.52	82.18	82.18
Credit	(16, 2, 690)	85.8	86.23	86.09	85.8
Diabetes	(9, 2, 768)	72.40	76.30	73.18	75.13
German	(21, 2, 1000)	70.90	73.40	73.00	74.5
Glass	(10, 7, 214)	67.29	76.17	77.57	78.04
Heart	(14, 2, 270)	80.00	81.48	80.74	83.33
Hepatitis	(20, 2, 155)	81.94	81.29	83.87	83.23
Horse	(23, 2, 368)	85.33	85.87	81.25	85.33
Ionosphere	(35, 2, 351)	89.17	92.02	93.73	93.45
Iris	(5, 3, 150)	94.67	94.67	94.00	93.33
Labor	(17, 2, 57)	78.95	84.21	89.47	84.21
Led7	(8, 10, 3200)	73.34	73.66	73.34	73.06
Lymphography	(19, 4, 148)	77.03	79.05	85.14	82.43
Pima	(9, 2, 768)	74.35	76.69	73.44	77.60
Sonar	(61, 2, 208)	78.85	78.85	84.62	85.58
Tic-tac-toe	(10, 2, 958)	83.72	93.84	98.54	95.82
Vehicle	(19, 4, 846)	71.04	74.11	78.25	74.94
Waveform	(22, 3, 5000)	76.44	83.30	83.90	84.04
Wine	(14, 3, 178)	94.38	96.07	97.75	97.75
Zoo	(17, 7, 101)	93.07	93.07	95.05	97.03

Gradient Boosting

Northeastern University

- Combination of boosting + gradient descent
- Each base model is trained to predict the gradients of the loss function w.r.t the predictions of the previous models
- Similar to gradient descent but operates in the function space
  - Known as **functional gradient descent**
- When the base models are decision trees, it is called gradient-boosted trees (GBDT)
- One of the best algorithms for structured data
- The original algorithm was developed by J.H. Friedman in 2001
  - Greedy function approximation: A gradient boosting machine
- Has many variants and enhancements
  - XGBoost, CatBoost, LightGBM, …

Basic Idea

Northeastern University

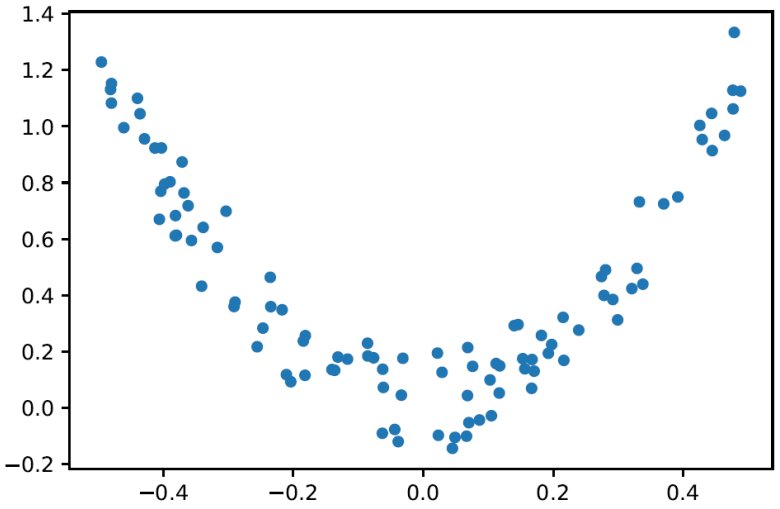
- Given a regression problem with a labeled data set (x_i, y_i) $(i = 1,.., n)$
- Build a sequence of trees where each tree predicts the residuals of the previous one
  - Fit a model $h_1(x)$ to the given labels y
  - Set the initial ensemble to $F_1(x) = h_1(x)$
  - Fit a model $h_2(x)$ to the residuals $y - F_1(x)$
  - Combine the two models: $F_2(x) = h_1(x) + h_2(x)$
  - Fit a model $h_3(x)$ to the residuals $y - F_2(x)$
  - Combine the three models: $F_3(x) = h_1(x) + h_2(x) + h_3(x)$
  - Continue for M steps
  - Return $F_M(x)$ as the final ensemble

Basic Idea

Northeastern University

- We can demonstrate the process in Python by manually building the sequence of regression trees
- Let's first generate a noisy quadratic training set:

```
n_samples = 100
X = np.random.rand(n_samples, 1) - 0.5
y = 5 * X[:, 0] ** 2 + 0.1 * np.random.randn(n_samples)
```



Basic Idea

Northeastern University

- We first fit a decision tree with max_depth=2 to the data set:

```
h1 = DecisionTreeRegressor(max_depth=2)
h1.fit(X, y)
```

```
DecisionTreeRegressor(max_depth=2)
```

```
F1 = [h1]  # ensemble of one tree
F1_pred = h1.predict(X)
print(f'R2 score of F1: {r2_score(y, F1_pred):.4f}')
```

```
R2 score of F1: 0.7819
```

- Now we train a second tree on the residual errors made by the first tree:

```
h2 = DecisionTreeRegressor(max_depth=2)
y2 = y - F1_pred
h2.fit(X, y2)
```

```
DecisionTreeRegressor(max_depth=2)
```

```
F2 = [h1, h2] # ensemble of two trees
F2_pred = sum(h.predict(X) for h in F2)
print(f'R2 score of F2: {r2_score(y, F2_pred):.4f}')
```

```
R2 score of F2: 0.8802
```

Basic Idea

Northeastern University

- And a third tree on the residual errors made by the previous trees:

```
h3 = DecisionTreeRegressor(max_depth=2)
y3 = y - F2_pred
h3.fit(X, y3)
```

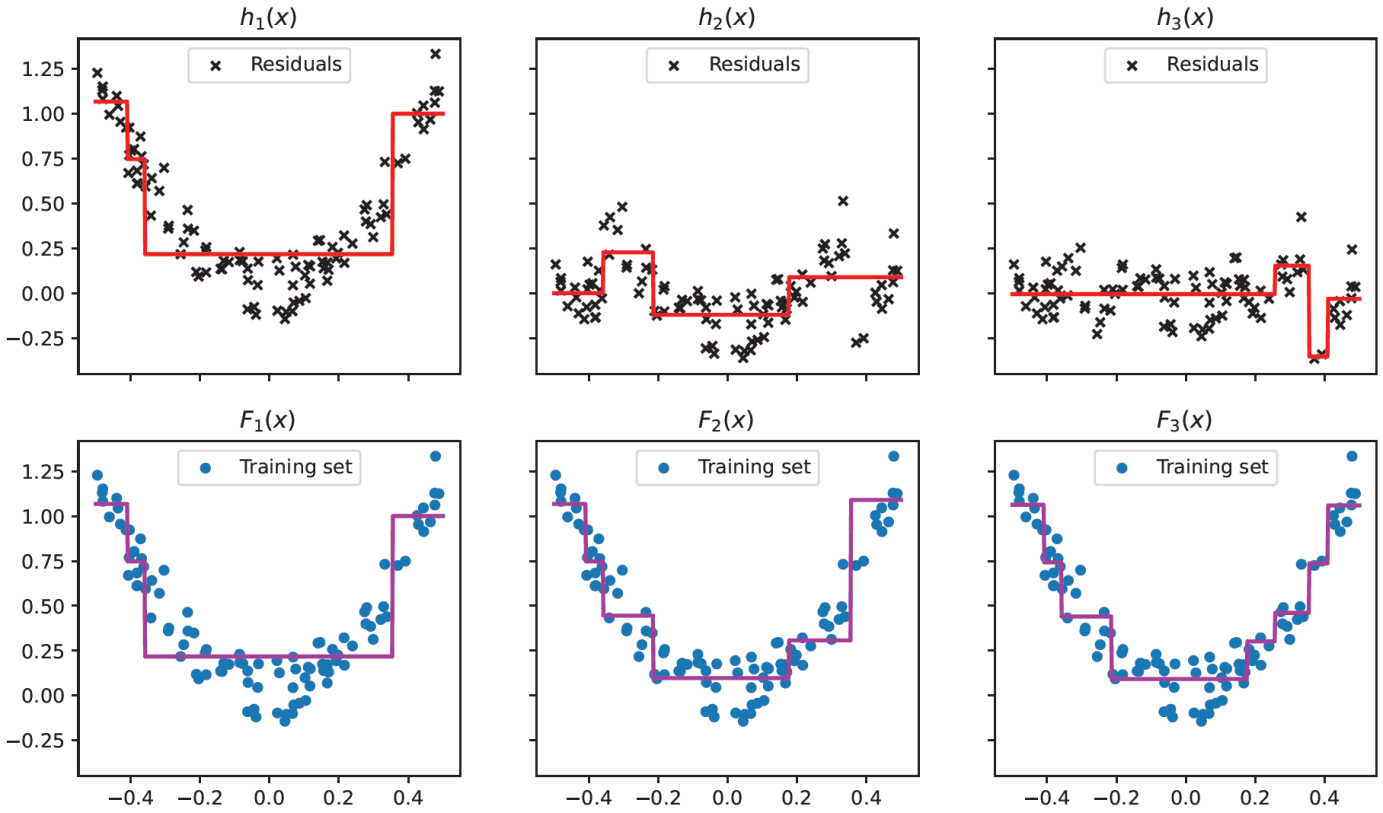
```
DecisionTreeRegressor(max_depth=2)
```

```
F3 = [h1, h2, h3] # ensemble of three trees
F3_pred = sum(h.predict(X) for h in F3)
print(f'R2 score of F3: {r2_score(y, F3_pred):.4f}')
```

```
R2 score of F3: 0.9124
```

Basic Idea

- The ensemble's predictions gradually get better as trees are added to the ensemble:



The Gradient Boosting Algorithm

Northeastern University

- Key observation: the residuals $h_m(\mathbf{x}_i)$ are proportional to the negative gradients of the squared loss function w.r.t the previous ensemble $F_{m-1}(\mathbf{x}_i)$:

$$L_{\text{squared}}(y_i, F_{m-1}(\mathbf{x}_i)) = (y_i - F_{m-1}(\mathbf{x}_i))^2$$

$$-\frac{\partial L_{\text{squared}}(y_i, F_{m-1}(\mathbf{x}_i))}{\partial F_{m-1}(\mathbf{x}_i)} = 2(y_i - F_{m-1}(\mathbf{x}_i)) = 2h_m(\mathbf{x}_i)$$

- We can generalize the algorithm to any differentiable loss function, by using the gradients of the loss function instead of the residuals

The Gradient Boosting Algorithm

Northeastern University

Algorithm Gradient Boosting

Input:

a training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$
a differentiable loss function $L(y, F(\mathbf{x}))$
a base learner algorithm
number of boosting iterations M

1: Initialize the model with a constant value:

$$F_0(\mathbf{x}) = \underset{\gamma}{\operatorname{argmin}} \sum_{i=1}^{n} L(y_i, \gamma)$$

2: **for** $m = 1$ to M **do**:
3: Compute the pseudo-residuals:

$$\tilde{y}_i = -\frac{\partial L(y_i, F_{m-1}(\mathbf{x}_i))}{\partial F_{m-1}(\mathbf{x}_i)}$$

4: Fit a base learner $h_m(\mathbf{x})$ to the training set $\{(\mathbf{x}_i, \tilde{y}_i)\}_{i=1}^n$
5: Update the model:

$$F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + h_m(\mathbf{x})$$

6: Output the final hypothesis $F_M(\mathbf{x})$

Regularization

Northeastern University

- Friedman suggested two regularization techniques
- **Shrinkage**: a learning rate *v* scales the contribution of each tree

$$F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \nu h_m(\mathbf{x})$$

  - Typically *v* <= 0.1
- **Subsampling**: the base learners are trained only on a fraction *f* of the training set
  - Drawn at random without replacement
  - *f* is typically set to 0.5

Gradient Boosting in Scikit-Learn

Northeastern University

- GradientBoostingRegressor is used for regression problems

class sklearn.ensemble.**GradientBoostingRegressor**(*, *loss='squared_error', learning_rate=0.1, n_estimators=100, subsample=1.0, criterion='friedman_mse', min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_depth=3, min_impurity_decrease=0.0, init=None, random_state=None, max_features=None, alpha=0.9, verbose=0, max_leaf_nodes=None, warm_start=False, validation_fraction=0.1, n_iter_no_change=None, tol=0.0001, ccp_alpha=0.0*) [source]

- GradientBoostingClassifier is used for classification problems

class sklearn.ensemble.**GradientBoostingClassifier**(*, *loss='log_loss', learning_rate=0.1, n_estimators=100, subsample=1.0, criterion='friedman_mse', min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_depth=3, min_impurity_decrease=0.0, init=None, random_state=None, max_features=None, verbose=0, max_leaf_nodes=None, warm_start=False, validation_fraction=0.1, n_iter_no_change=None, tol=0.0001, ccp_alpha=0.0*) [source]

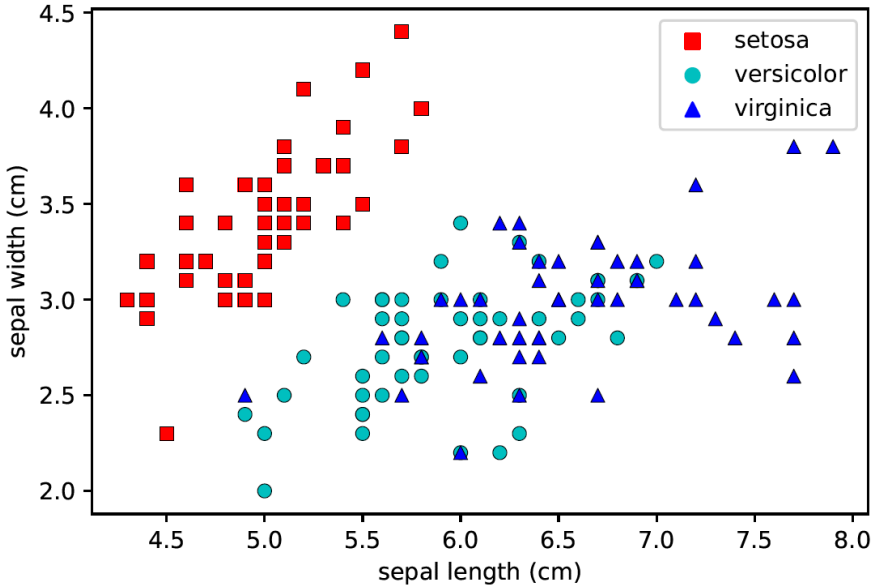
- The hyperparameters control the size of the trees (e.g., max_depth), the number of trees and the learning rate

Classification Example

Northeastern University

- Let's train a gradient boosting classifier on the Iris data set
  - Using only the first two features of each flower (sepal width and sepal length)

```
from sklearn.datasets import load_iris

iris = load_iris()
X = iris.data[:, :2] # we only take the first two features
y = iris.target
```



- Splitting to 75% training and 25% test:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
```

Classification Example

Northeastern University

- Training a gradient boosting classifier with its default settings
  - i.e., an ensemble of 100 trees with max_depth=3

```
from sklearn.ensemble import GradientBoostingClassifier

clf = GradientBoostingClassifier(random_state=42)
clf.fit(X_train, y_train)
```

```
GradientBoostingClassifier(random_state=42)
```

```
print(f'Train accuracy: {clf.score(X_train, y_train):.4f}')
print(f'Test accuracy: {clf.score(X_test, y_test):.4f}')
```

```
Train accuracy: 0.9554
Test accuracy: 0.7895
```

Classification Example

Northeastern University

- Tuning the hyperparameters of the model:

```
from sklearn.model_selection import RandomizedSearchCV

params = {
    'n_estimators': [10, 50, 100, 200, 500],
    'max_depth': np.arange(3, 11),
    'subsample': np.arange(0.5, 1.0, 0.1),
    'max_features': ['sqrt', 'log2', None]
}
search = RandomizedSearchCV(GradientBoostingClassifier(random_state=42), params,
                            n_iter=50, cv=3, n_jobs=-1)
search.fit(X_train, y_train)

print(search.best_params_)
```

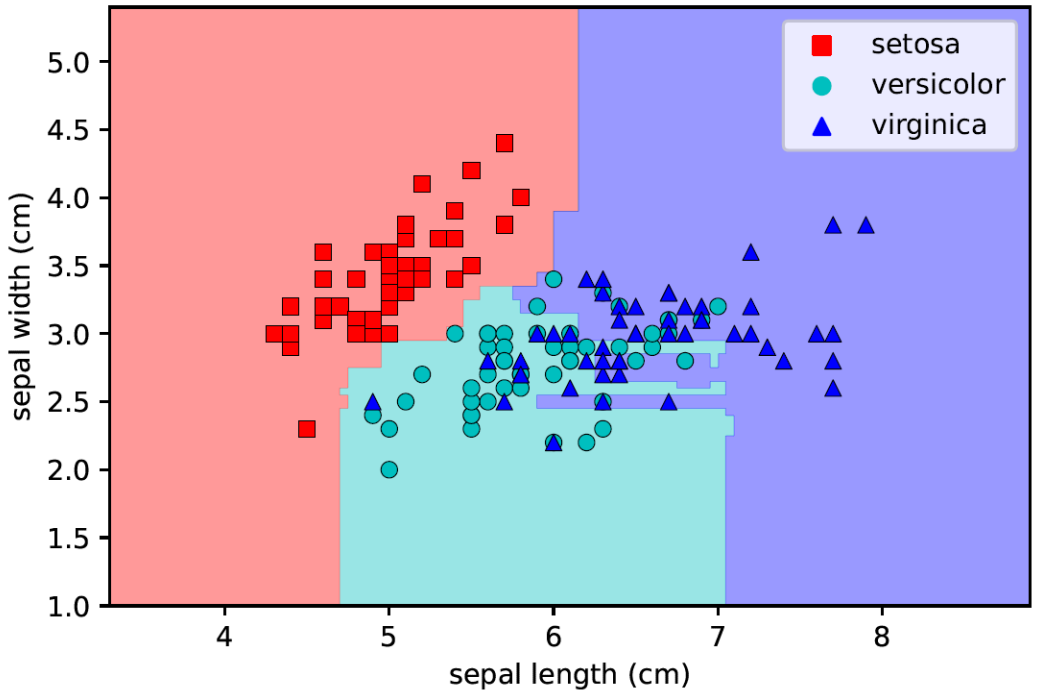
```
{'subsample': 0.6, 'n_estimators': 10, 'max_features': 'sqrt', 'max_depth': 3}
```

```
best_clf = search.best_estimator_
print(f'Train accuracy: {best_clf.score(X_train, y_train):.4f}')
print(f'Test accuracy: {best_clf.score(X_test, y_test):.4f}')
```

```
Train accuracy: 0.8125
Test accuracy: 0.8684
```

Classification Example

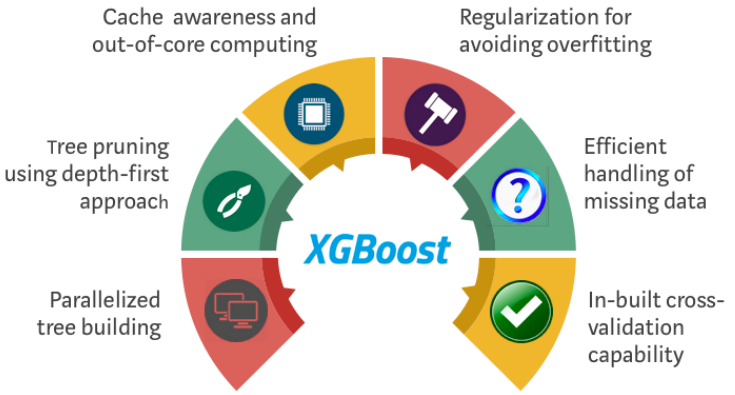
- The decision boundaries of the best model:



XGBoost (eXtreme Gradient Boosting)

Northeastern University

- An optimized distributed gradient boosting library
- Provides state-of-the-art results on many real-world data sets with structured data
- Can solve problems with billions of examples
- Can run on various distributed environments (e.g., Hadoop, AWS, Azure)
- Provides Scikit-Learn wrapper interfaces for its native classes
- Documentation of the library can be found at https://xgboost.readthedocs.io/
- Installation: pip install xgboost

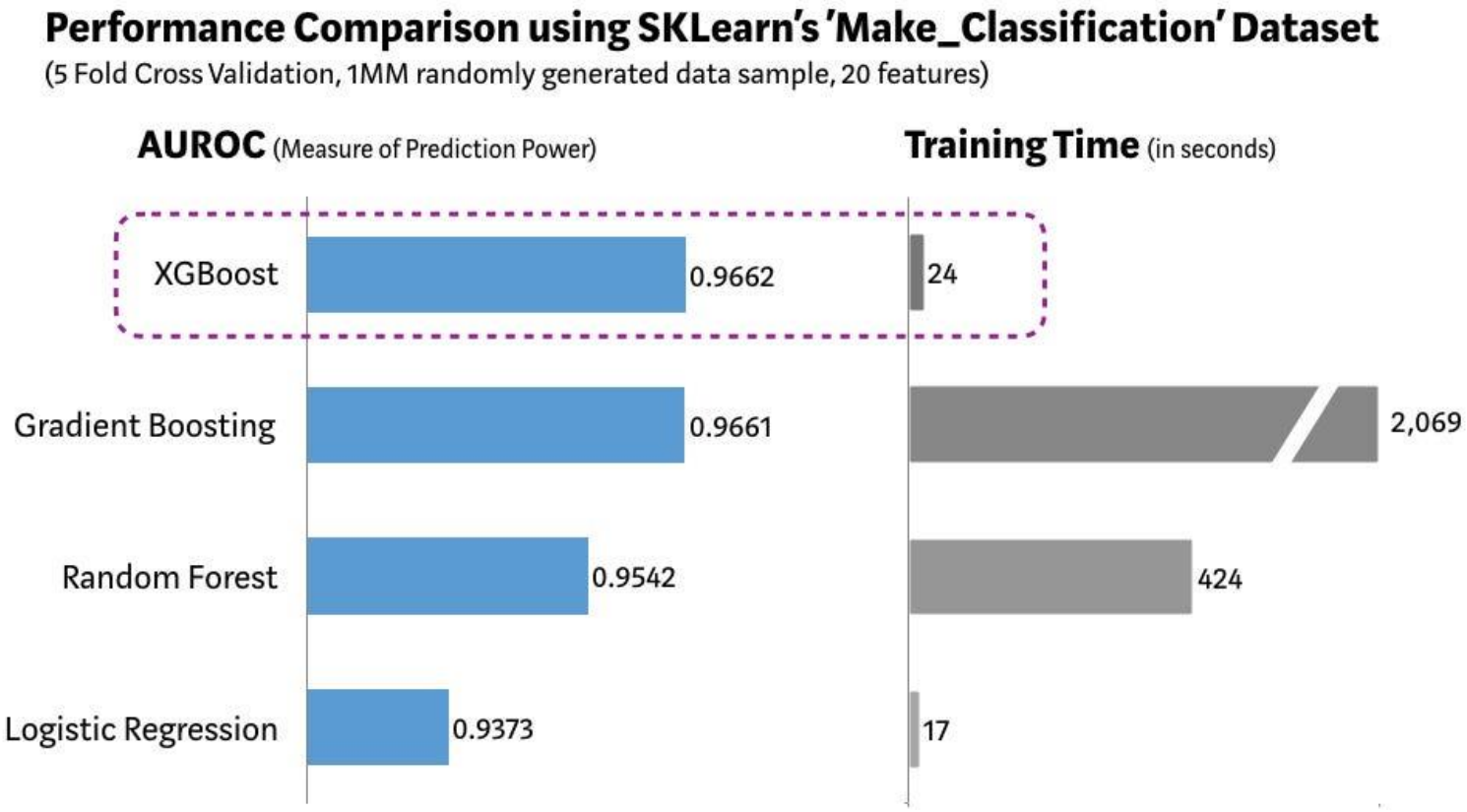


XGBoost Features

Northeastern University

- Use Newton's method in the function space instead of gradient descent
- Incorporates clever regularization techniques to penalize complex trees
- Uses a novel tree learning algorithm for handling sparse data
- Supports feature subsampling for each tree / tree level / tree node
- Uses theoretically justified quantile sketching for efficient computation
- Can handle missing values without imputation
- Supports parallel processing by distributing the workload across multiple threads
- Supports out-of-core computation using an efficient cacheable block structure
- Supports distributed computation
- Supports cross-validation
- Can handle wide range of problems, including classification, regression and ranking

XGBoost Features



XGBClassifier

Northeastern University

class **xgboost.XGBClassifier**(*, *objective='binary:logistic', use_label_encoder=None, **kwargs*)

Parameter	Description
n_estimators	Number of boosting rounds (defaults to 100)
max_depth	Maximum tree depth for base learners (defaults to 6)
learning_rate	Boosting learning rate (defaults to 0.3)
objective	Specify the learning task and the corresponding learning objective
n_jobs	Number of parallel threads used to run xgboost
gamma	Minimum loss reduction required to make a further partition on a leaf node of the tree
subsample	Subsample ratio of the training set
reg_alpha	L1 regularization penalty term on weights
reg_lambda	L2 regularization penalty term on weights
scale_pos_weight	Balancing of positive and negative weights
early_stopping_rounds	Activates early stopping

XGBRegressor

Northeastern University

```
class xgboost.XGBRegressor(*, objective='reg:squarederror', **kwargs)
```

- Has similar parameters to XGBClassifier

Classification Example

Northeastern University

- Training an XGBoostClassifier on the Iris data set:

```
from xgboost import XGBClassifier

clf = XGBClassifier(random_state=42, max_depth=3, learning_rate=0.1)
clf.fit(X_train, y_train)
```

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, gpu_id=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=0.1, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=3, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
              n_estimators=100, n_jobs=None, num_parallel_tree=None,
              objective='multi:softprob', predictor=None, ...)
```

```
print(f'Train accuracy: {clf.score(X_train, y_train):.4f}')
print(f'Test accuracy: {clf.score(X_test, y_test):.4f}')
```

```
Train accuracy: 0.8839
Test accuracy: 0.7895
```

Gradient Boosting Summary

Northeastern University

Pros

- Provides highly accurate models
- Among the best models that exist today for structured data
- Can effectively handle high-dimensional data sets
- Less sensitive to outliers
- Can handle various data types
- Can handle missing values
- Provides feature importance
- Supports a wide range of loss functions

Cons

- Training can be computationally intensive if the number of trees is large
  - Building of the trees cannot be parallelized
- Less interpretable than single decision trees
- Can overfit the training set if not properly regularized
- Several hyperparameters need to be tuned (tree size, learning rate, number of trees)